# A DATA-DRIVEN ARCHITECTURE FOR RAPID PROTOTYPING OF HIGH THROUGHPUT DSP ALGORITHMS

**Alfred K.W. Yeung and Jan M. Rabaey**
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720.

**Abstract: A data-driven multiprocessor architecture for rapid prototyping of complex DSP algorithms, based on direct execution of data-flow graphs, is presented. High computation bandwidth is achieved by exploiting fine-grain parallelism inherent in the target algorithms using simple processing elements interconnected by a flexible static communication network. The use of distributed control and data-driven principle of execution results in a highly scalable and modular architecture. A prototype chip, which is being designed, will contain 64 nanoprocessors and provide 3.2 GOPS running at 50 MHz. The benchmark results based on a variety of DSP algorithms in video processing, digital communication, digital filtering and speech recognition confirm the performance, efficiency and generality of the architecture.**

## 1. Introduction

During the development of real-time, complex and high performance DSP algorithms, a hardware prototype of the designed system plays a very important role in shortening design time and in improving design quality. A prototype allows the designer to quickly explore a large design space, to fine-tune algorithm parameters and to verify and evaluate the performance of the algorithms under realistic conditions, e.g., to experiment with a video compression algorithm while transmitting over a noisy communication channel, all in real-time.

Given today's technology constraints, a single-chip solution cannot satisfy the high computation requirement. The traditional approach of putting together a system using existing chips requires dedicated interface design and depends on the availability of chips with the desired functionality. Moreover, compilation is more difficult for a heterogeneous hardware platform.

Multiprocessing using general purpose DSP processors, though very flexible, does not offer a cost-effective solution because the Von Neumann processors, which invest heavily on control functionalities to enable time-multiplexing operations on a few execution units, is not efficient (or too complex) for exploiting fine grain parallelism in complex algorithms which offer limited opportunity for hardware sharing.

Systolic or wavefront array processors achieve high performance using regular, fine grain, locally interconnected processing elements [1]. However, the technique is only applicable for a certain class of algorithms [2] and specially designed I/O units are often needed to provide high I/O bandwidth to sustain the high throughput [3].

Field-programmable Gate-arrays (FPGA), e.g., Xilinx [4], are extremely flexible by providing programmability down to the gate level but their bit-oriented

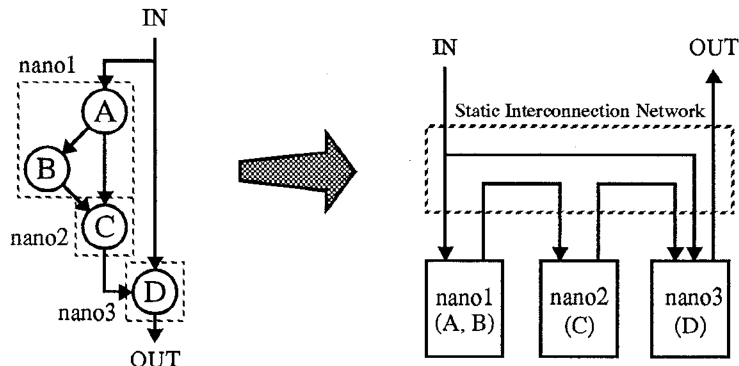



Figure 1. Direct Execution of Data-flow graph using Nanoprocessors interconnected statically.

architectures are inefficient in terms of area and are slow in implementing datapath intensive architectures commonly used in high throughput DSP algorithms.

PADDI, a reconfigurable multiprocessor architecture specially developed for prototyping of high speed datapaths, demonstrated the efficiency of exploiting fine grain parallelism using simple execution units [5][7]. However, the architecture may run into instruction bandwidth problems and the use of a global controller limits the scalability of the architecture.

In this paper, a flexible and scalable multiprocessor architecture geared towards rapid prototyping of high sampling rate, high throughput DSP algorithms with fine grain parallelism is described.

## 2. Data-driven DSP Architecture

In real-time, high throughput and high sampling rate DSP algorithms, operations in the kernels (inner loops) often need to be processed in a small number of clock cycles, implying limited opportunity for hardware sharing [6]. Researchers have found that an efficient way to exploit the fine grain parallelism in this type of algorithms is to use a network of simple processing elements, whose connection pattern closely matches the data flow pattern of the algorithm [7][8].

Moreover, very simple controllers are often sufficient because of the dedicated nature of the processing elements. This implies that in the data-flow graph representing the target algorithms, the majority of the nodes perform useful work and few nodes are needed for control or data routing. If the data-flow graphs of such algorithms are implemented by assigning a processor element to a node or a cluster of a small number of nodes, the resulting architecture would be efficient in the sense that the number of overhead processing elements is small. An extreme case is a systolic algorithm whose data-flow graph has no overhead node.

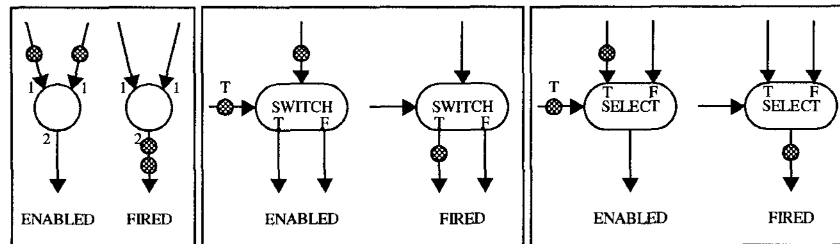
This motivates us to investigate a multiprocessor architecture which directly executes a data-flow graph by mapping the graph onto simple processing elements, called *nanoprocessors*, connected by a static interconnection network. As illustrated in Figure 1, a nanoprocessor is simply a hardware realization of a node or an encapsulation of several nodes. Like their data-flow counterparts, nanoprocessors communicate with each other via static communication channels (arcs in the graph)

using streams. Execution of an operation (firing of a node) by a nanoprocessor is data-driven, i.e., execution depends on the availability of the required operands. In essence, the architecture is very much like a message-passing multicomputer system, which is considered to be the most scalable multicomputer system of all [9], except that very simple computers and static communication are used.

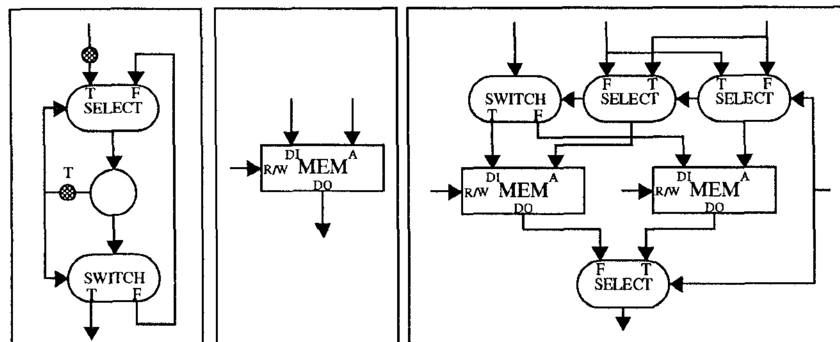
It has been widely accepted that data-flow graph is a natural, perhaps the best, representation for signal processing algorithms [10] because it exposes all parallelism in the algorithms and real signals are naturally represented by the data streams implied in data-flow. Because the architecture directly supports the semantics of data-flow, the task of developing an efficient compiler, which is crucial for rapid prototyping, is greatly simplified.

## 2.1 Static Data-flow

In this section, the data-flow model used for describing DSP algorithms is explained. Among the many variations of data-flow models [11], we adopt Dennis' static data-flow model [12]. A data-flow program is described by a directed graph where the nodes denote operations, e.g., addition, and the arcs denote data dependencies. Dynamic behaviors due to data-dependent iterations and conditionals are handled by the SWITCH and SELECT operators. Figure 2a gives examples of the basic data-flow operators[1] and a data-flow graph implementing a while-loop is shown in Figure 2b.




(a) Examples of Data-flow Operators.




(b) While-loop. (c) Memory Node. (d) Swapping Memory Banks.

Figure 2. Static Data-flow Model.

1. The number next to the arcs denotes the number of tokens consumed or produced on each firing and is omitted if the number is one for clarification.

The arcs going in and out of the nodes represent physical connections called *channels* between the nanoprocessors. To make the implementation of channels practical, the First-in-first-out (FIFO) schema is enforced on the arcs, i.e., the arcs are simply FIFO queues of bounded capacity.

Any algorithm can be described by using only the basic data-flow nodes shown in Figure 2a. Figure 3(a) shows the *minimal nanoprocessor* which can implement these basic data-flow nodes: by interconnecting multiple minimal nanoprocessors, *any* algorithm can be implemented. However, the implementation is most efficient for algorithms which are computation intensive and require simple control. These are precisely the characteristics of our target algorithms.

A *pure* data-flow node does not have internal state. By allowing internal states, a node can be extended to represent an encapsulation of several nodes.A cluster of nodes simply denotes a number of primitive operations executed in a specific order. Thus data-flow nodes can be implemented either spatially by a network of nanoprocessors or temporally by execution of a sequence of instructions by a single nanoprocessor. In the later case, the nanoprocessor functions like a traditional Von Neumann machine. To support the time multiplexing of several nodes onto a single nanoprocessor, a small program store is added to the minimal nanoprocessor as shown in Figure 3(b).

By allowing internal states in data-flow nodes, memory can be easily incorporated into the data-flow model (Figure 2c), preserving a consistent programming paradigm across the architecture. A memory node is similar to other data-flow nodes except that it has a very large number of internal states. It consumes an address token and a control token specifying read or write operation. For a read operation, it generates an output token containing data stored at the location

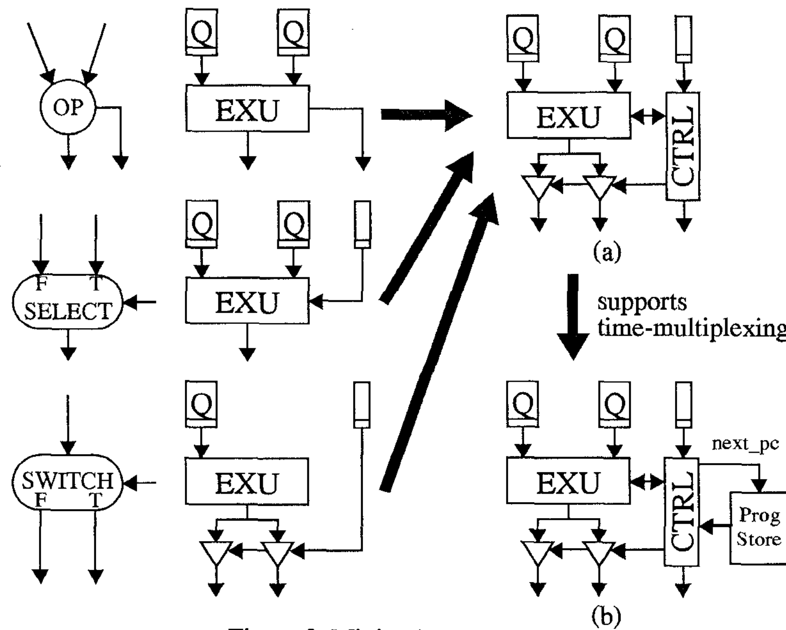



Figure 3. Minimal Nanoprocessor.

pointed at by the address token. For a write operation, it consumes an additional input data token and simply updates its internal states by storing the data. Memory nodes, together with the SWITCH and SELECT operators, can create many powerful shared memory structures. An example of swapping memory banks often used in video processing to buffer successive pixel frames is shown in Figure 2d.

## 2.2 Simple Homogeneous Processing Element: Nanoprocessor

Due to limited opportunity for hardware sharing and high computation requirement, a nanoprocessor is often dedicated to the processing of only a small number of operations, and therefore a very simple and homogeneous nanoprocessor has been adopted for the architecture. The simplicity of the nanoprocessor results in high integration and fast cycle time. Although heterogeneous processing elements may be more efficient in terms of hardware costs for some applications, the additional resource types are likely to complicate the compilation process. Moreover, the regularity also simplifies interconnection network design and improves layout density. More complex functions can be built by hooking up more nanoprocessors.

The data-flow principle implies distributed control, i.e., each nanoprocessor is equipped with its own local controller. Execution of a program in one nanoprocessor has no *side-effect* on the others. By eliminating global control and side-effect, the scalability and modularity of the architecture is ensured. The local controller relieves instruction bandwidth bottleneck and allows the use of long and fully decoded instructions, resulting in fast cycle time and a powerful instruction set. Thanks to the local controller, zero-cycle branch latency as well as multi-way branch can be achieved with very little speed penalty.

Nanoprocessors communicate with each other explicitly using data and control streams in a data-driven manner. The data-driven principle of execution gracefully synchronizes a large number of nanoprocessors typically needed for the target algorithms. The data-driven handshakes provide a uniform interface between *all* elements of the architecture and improve the scalability and modularity of the architecture.

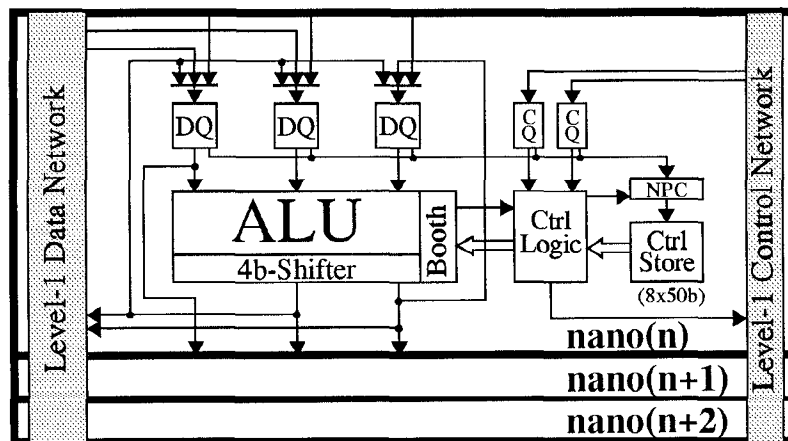



Figure 4. Nanoprocessor Block Diagram.

Figure 4 shows the block diagram of a nanoprocessor. It contains an 8-word by 50-bit instruction store, a 16-bit ALU which performs arithmetic and logic operations with special support for a modified Booth multiplication step, a 4-bit shifter and one 4-word data buffer for each of the three inputs. The instruction store is initialized at system start-up using a serial interface. The data buffer can be configured as a simple data queue or as a small random-access register file. Data at the head of the queue can be held for reuse. The registers are scannable so constants or filter coefficients can be loaded at start-up. All instructions take one cycle to complete.

The local controller maintains a 3-bit program counter. A next program counter field (NPC) is available in the instruction word. Bits of the NPC can be modified depending on the status of the ALU or the token at the head of the control queue to implement multi-way branch with zero-cycle latency. The nanoprocessor is stalled if any of the required operands (data or control tokens) are not available or if any of the output channels are blocked.

Inputs of the two data buffers are connected to the Level-1 network while only neighboring connection is allowed for the third. The third input is specifically provided to allow pipeline implementation of a 16b x 16b multiplier using a single nano-cluster consisting of 4 nanoprocessors. In order to implement the SWITCH node, two output connections are available to the Level-1 network. To facilitate neighboring communication and reduce network bandwidth demand, direct connections between neighbors are implemented without access to the Level-1 network. Similarly, two input channels and one output channel are available for the control streams.

## 2.3 Static Hierarchical Interconnection Network

In an effort to reduce area overhead caused by interconnection networks, many array architectures provide rather restrictive interconnection networks [13]. This limits the scope of the application, e.g., to systolic algorithms only, and may degrade performance for algorithms with recursions. Moreover, the lack of a flexible network further increases the burden on the compiler to generate a good processor placement scheme. To overcome the aforementioned inefficiencies, the architecture provides a flexible communication network to match the communication patterns of a wide range of algorithms and to maximize nanoprocessor usage.

Motivated by the static interconnection used in connecting nodes in data-flow graphs, the architecture provides only static point-to-point communications between nanoprocessors. This greatly simplifies network design and eliminates the needs for sophisticated network routing hardware commonly called for in general multicomputer systems [9]. Speed of the network can also be improved potentially because the large transistors typically used in implementing switches in the network do not have to be switched every cycle.

The lack of dynamic interconnection does not impose serious restrictions. Because nanoprocessors are usually dedicated to the processing of a few operations, channels are likely to be saturated with data and therefore the opportunity for channel sharing is very limited in the first place. A study by Chen [7] on the interconnection patterns of a set of high throughput, high sampling rate DSP algorithms show that 95% of the communication channels are either not shared or shared by fewer than two clients. Second, dynamic interconnection patterns can be emulated by programming the nanoprocessors to implement the SWITCH and SELECT nodes.

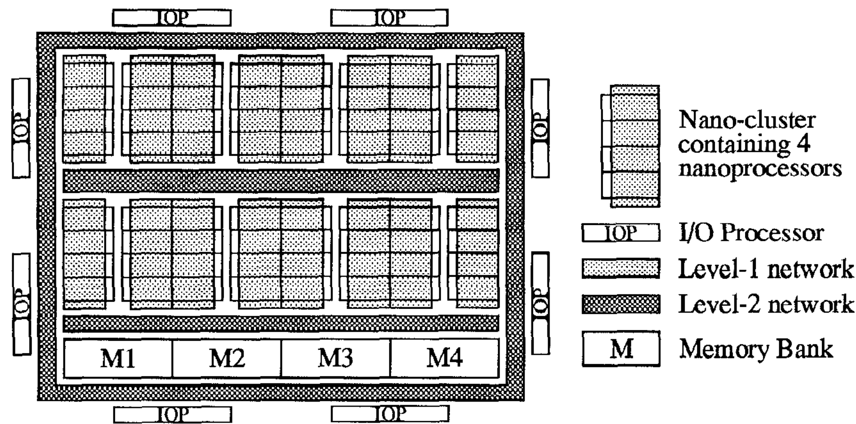



Figure 5a. Block Diagram of the prototype chip.

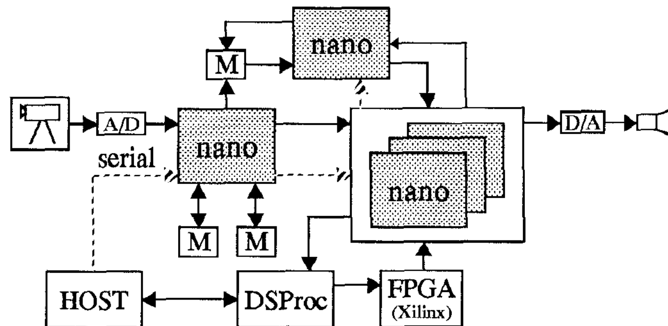



Figure 5b. Example of Prototype DSP system.

To trade off communication latency with area overhead and cycle time, a hierarchical pipeline network is used. As shown in the chip block diagram in Figure 5a, the level-1 network is layout directly on top of a 4-nanoprocessor cluster as datapath feed-throughs to save area. Similar to the interconnect matrix design in [14], a semi-crossbar with switchable buses is used to implement the level-2 network to reduce area cost by exploiting the locality of communication.

Two lines (*send* and *busy*) are associated with each channel to implement the handshake protocol. Data broadcasting, which is used heavily for distributing data when exploiting spatial concurrency, is efficiently implemented using wired-logic for the *busy* signal.

## 2.4 Overall Chip Architecture

A block diagram showing the prototype chip architecture is shown in Figure 5a. Each chip contains 16 nano-clusters organized in 2 banks and four on-chip memory banks (4 x 128 words x 16 bits) interconnected via a 2-level configurable pipeline network. Each nano-cluster consists of 4 nanoprocessors and a set of pipeline registers between the level-1 and level-2 interconnection networks. Eight 16-bit I/O ports, each controlled by a specialized I/O nanoprocessor called IOP, provide high I/O bandwidth needed for sustaining the high throughput. Two ports can be linked

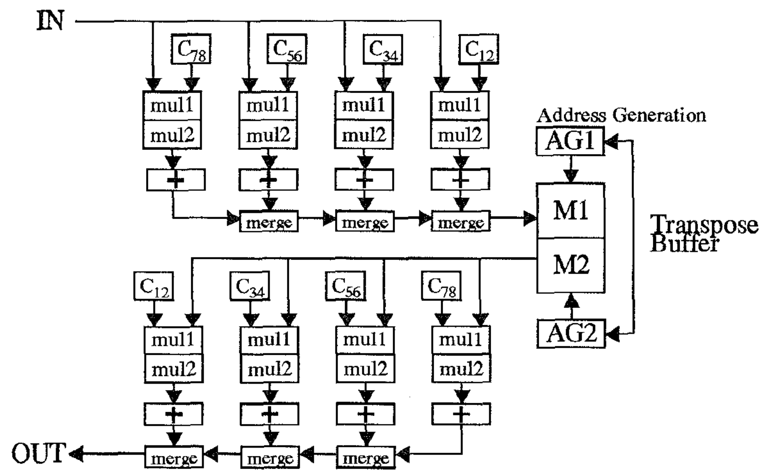



Figure 6. 2-D 8x8 DCT at 25 MHz sampling rate using 50 Nanoprocessors.

to provide 32-bit I/O if necessary. The primary function of the IOP is to control off-chip communication with another nanoprocessor chip or memory chip. A serial port handles the downloading of nanoprocessor instructions, network configuration and data buffer initialization. Selective internal scan chains and a boundary scan chain are provided for system testing and run-time debugging support. For example, a nanoprocessor can be programmed to collect relevant system statistics such as how often an arithmetic overflow occurs at a certain node of the algorithm. An example of a DSP system prototyped by the nanoprocessor array is shown in Figure 5b.

A prototype chip of the architecture, which is being designed, can provide 3.2 GOPS, 400 MB/sec on-chip memory bandwidth and 800 MB/sec I/O bandwidth at the estimated clock frequency of 50 MHz.

## 3. Benchmark Results

A benchmark set consisting of algorithms in the areas of video processing, speech recognition, digital communication and digital filtering was analyzed to justify the architectural choices and verify the performance and generality of the architecture. Figure 6 shows the implementation of a 2-dimensional 8x8 Discrete Cosine Transform (DCT) at 25 MHz sampling rate using 50 nanoprocessors and two on-chip memory banks used as transpose buffers. The algorithm is decomposed into two one-dimensional DCT's and a transpose operation. Multiplication is implemented in a pipeline fashion using two nanoprocessors. The transform coefficients stored in the data buffers of the nanoprocessors are streamed into the multiplication nanoprocessors together with the input pixels. Address generation unit (AG2), which is very similar to the 2-dimensional space address generator in algorithm 5 in Table 1, produces the desired address sequence for the transpose operation.

Table 1 lists the number of core operations per data sample (not counting overhead), the performance achieved and the amount of on-chip hardware used for the benchmark algorithms. Most algorithms require irregular communications between nanoprocessors. The results of algorithms 1, 2 and 3 show good correlation

between performance gain and hardware requirement. The Viterbi decoder in 4 contains a tight recursive loop which limits performance. The normalized throughput per nanoprocessor chip averaged over the benchmark set is 1612 MOPS.

As a comparison, a general purpose DSP processor such as the TMS320C25 can only achieve a sampling rate of 250 KHz for the 2-dimensional 8x8 DCT with optimized code [15]. Interleaving several processors to achieve higher performance requires careful communication network design to overlap communication with computation.

**Table 1: Benchmark Results. (50 MHz estimated clock frequency)**

| # | Algorithms | Core Operations[a] (per sample) | Sampling-Rate | Throughput[b] | Hardware |
|---|---|---|---|---|---|
| 1 | 2nd-order IIR filter [16] (original) | 5 mult, 4 add | 8.3 MHz | 75 MOPS | 6 nano |
| 2 | 2nd-order IIR filter [16] (pipeline and multiplexed) | 10 mult, 11 add | 25 MHz | 525 MOPS | 27 nano |
| 3 | 2nd-order IIR filter [16] (pipeline) | 10 mult, 11 add | 50 MHz | 1050 MOPS | 54 nano |
| 4 | Viterbi ACS unit [17] | 15 comp & select | 5.6 MHz (9 Mb/s) | 168 MOPS | 10 nano |
| 5 | 2-dim space address generator [19] | 2 add, 1 comp | 50 MHz | 150 MOPS | 3 nano |
| 6 | 3x3 sorting filter | 15 comp & select | 50 MHz | 1500 MOPS | 30 nano |
| 7 | Video Matrix conversion | 4 mult, 6 add (8b pixel) | 50 MHz | 500 MOPS | 14 nano |
| 8 | 2-dim 8x8 DCT (multiplexed) | 16 mult-add (8b pixel) | 25 MHz | 800 MOPS | 50 nano, 2 mem |
| 9 | 2-dim 8x8 DCT (non-multiplexed) | 16 mult-add (8b pixel) | 50 MHz | 1600 MOPS | 82 nano, 2 mem |
| 10 | Viterbi Word Processor [18] | 5 add, 3 comp, 3 select, 3 mem rd & wr | 3 state updates at 25 MHz | 275 MOPS | 13 nano, 3 mem |
| 11 | Motion Vector Estimation 8x8 blk, 16x16 search blk | 64 sub, comp & add | 22.2 MHz | 4267 MOPS | 4 nano-chips |

a. All operations are 16 bit unless specified otherwise.
b. Multiplication is counted as 1 operation.

## 4. Conclusion

A data-driven multiprocessor architecture for rapid prototyping of complex DSP algorithms, based on direct execution of data-flow graphs, is presented. High computation bandwidth is achieved by exploiting fine-grain parallelism inherent in the target algorithms using simple nanoprocessors. The powerful hierarchical network provides flexibility to accommodate a wide range of algorithms. The distributed control strategy and the data-driven principle of execution result in a highly scalable and modular architecture that can be easily extended to solve more complex problems. The simplicity and homogeneity of the architecture, the consistent programming paradigm and the direct architectural supports for the execution of data-flow graphs facilitate development of compiler and synthesis tools needed for rapid prototyping.

The benchmark results presented confirm the performance, efficiency and generality of the architecture. Running at the estimated frequency of 50 MHz, the prototype chip consisting of 64 nanoprocessors, 2KB of data memory and eight 16-bit I/O ports can provide 3.2 GOPS, 400 MB/sec on-chip memory bandwidth and 800 MB/sec I/O bandwidth.

## 5. Acknowledgments

This research was funded by DARPA.

## 6. References


[1] S. Y. Kung, "On supercomputing with systolic/wavefront array processors," Proceedings of the IEEE, pp. 39-46, 1984.

[2] S. K. Rao, Regular iterative algorithms and their implementation on processor arrays. Ph.D. thesis, Stanford University, 1985.

[3] H. Volkers et al.. "Cache Memory Design for the Data Transport to Array Processors". ICASSP, 1990.

[4] R. Freeman. "User-programmable Gate Arrays". IEEE Spectrum, Dec. 1988.

[5] D. Chen and J. Rabaey. "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real-time Data Paths", ISSCC, 1992.

[6] Rabaey, C. Chu, P. Hoang, M. Potkonjak, "Fast Prototyping of Data Path Intensive Architecture", IEEE Design and Test, pp.40-51, 1991.

[7] D. Chen, "Programmable Arithmetic Devices for High Speed Digital Signal Processing", PhD thesis, University of California at Berkeley, May, 1992.

[8] W. Geurts, F. Catthoor, "DSP Applications suited for Lowly Multiplexed Architectures", ASICS Open Workshop on Synthesis Techniques for (Lowly) Multiplexed Datapaths, Aug. 1990.

[9] S. Borkar et al.. "iWarp: An Integrated Solution to High-Speed Parallel Computing", Proceedings Supercomputing, Nov 1988.

[10] E. A. Lee, "Consistency in Dataflow Graphs", IEEE Transactions on Parallel and Distributed Systems, April, 1991.

[11] Arvind, D. Culler, "Dataflow Architectures", Annual Reviews in Computer Science, MIT, Laboratory for Computer Science, 1986.

[12] J. B. Dennis, "First Version Data Flow Procedure Language", Technical Memo MAC TM61, May 1975, MIT Laboratory for Computer Science.

[13] I. Koren, B. Mendelson, "A Data-driven VLSI Array for Arbitrary Algorithms", Computer Magazine, Oct., 1988.

[14] J. Wawrznyek, "A Reconfigurable Concurrent VLSI Architecture for Sound Synthesis", VLSI Signal Processing II, Nov. 1986.

[15] P. Papamichalis, "Digital Signal Processing Applications with the TMS320 Family, Vol. 3", Prentice Hall.

[16] K. Parhi and D. Messerschmitt. "Pipeline Interleaving and Parallelism in Recursive Digital Filters - Part I: Pipelining using Scattered Look-ahead and Decomposition", IEEE Transactions on ASSP, p 1109, vol.37, July 1989.

[17] C. Shung et al.. "A 30 MHz Trellis CODEC Chip for Partial-Response Channels", ISSCC, 1991.

[18] J. Rabaey et al., "A Large Vocabulary Real-Time Continuous Speech Recognition System", VLSI Signal Processing III, pp. 62-74, IEEE Press, Nov., 1989.

[19] M. Toyokura et al.. "A Video Digital Signal Processor with a Vector-Pipeline Architecture", ISSCC, 1992.